Parallel Computing Final-Term: *Mean Shift*

Giovanni Bindi

Università degli Studi di Firenze

24 February 2021

Introduction: *Mean Shift*

Mean shift is a popular non-parametric clustering technique developed during the 1970s [1]. It aims to locate the *modes* of a density function. Given a set of observations $\mathcal{S} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ with $\mathbf{x} \in \mathbb{R}^D$ and a kernel function $K_h : \mathbb{R}^D \to \mathbb{R}$ it operates a kernel density estimation:

$$f(\mathbf{x}) = \frac{1}{Nh^D} \sum_{i=1}^{N} K_h\Big(\frac{\mathbf{x} - \mathbf{x}_i}{h}\Big) \tag{1}$$

where $h \in \mathbb{R}$ is called the window radius (or width).

Idea

The modes of the density function are located at the zeros of the gradient $\nabla f(\mathbf{x}) \to$ derive an iterative algorithm for clustering [2].

Algorithm

Define the the weighted mean of the density, determined by K_h as

$$m(\mathbf{x}) = \frac{\sum_{\mathbf{x}_i \in N(\mathbf{x})} K_h(\mathbf{x} - \mathbf{x}_i)\mathbf{x}_i}{\sum_{\mathbf{x}_i \in N(\mathbf{x})} K_h(\mathbf{x} - \mathbf{x}_i)} \tag{2}$$

where $N(\mathbf{x})$ is a neighborhood of $\mathbf{x}$. Typically a gaussian kernel is used

$$K_\sigma(\mathbf{x} - \mathbf{x}_i) = e^{\frac{\|\mathbf{x} - \mathbf{x}_i\|}{2\sigma^2}} \tag{3}$$

For $t = 1, \ldots, I$ iterations, being $\mathcal{S}^{(1)} = \mathcal{S} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$:

1. **Calculate** $m(\mathbf{x}^{(t)})$, for each $\mathbf{x}^{(t)} \in \mathcal{S}^{(t)}$
2. **Update** $\mathbf{x}^{(t+1)} = m(\mathbf{x}^{(t)})$, for each $\mathbf{x}^{(t)} \in \mathcal{S}^{(t)}$

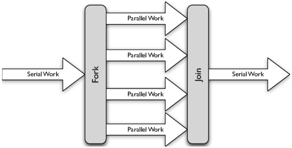
At the end reduce $\mathcal{S}^{(I)}$ to a set of clusters with centers $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$.

Pseudo-code: Sequential Version

Input: $\mathcal{S} \subset \mathbb{R}^D$. $I \in \mathbb{N}$. $\sigma, R, \delta \in \mathbb{R}$.
Output: Centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$

for $t = 1, \dots, I$ **do**
    for $i = 1, \dots, N$ **do**
        $\boldsymbol{\tau} = \mathbf{0}$
        $\eta = 0$
        for $j = 1, \dots, N$ **do**
            if $\|\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)}\|_2^2 \leq R$ **then**
                $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})\mathbf{x}_j^{(t)}$
                $\eta = \eta + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})$
            end if
        end for
        $\mathbf{x}_i^{(t+1)} = \boldsymbol{\tau}/\eta$
    end for
end for
$\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\} \leftarrow$ `reduce_centroids`$(\mathcal{S}^{(I)}, \delta)$
return $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$

OpenMP [3]



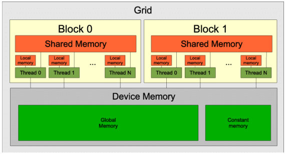
- Take advantage of **fork-join** via OpenMP.
- Only **few** directives needed for this task:
  - `#pragma omp parallel for`
  - `#pragma omp critical`
- Benefit from both **shared** and **private** memory.
- Exploit both workload sharing mechanism:
  - **Static** (set by the user).
  - **Dynamic** (set by OpenMP).

OpenMP: Pseudo-code

Input: $\mathcal{S} \subset \mathbb{R}^D$. $I \in \mathbb{N}$. $\sigma, R, \delta \in \mathbb{R}$.
Input: Optional: number of threads $T \in \mathbb{N}$.
Output: Centroids $\{\boldsymbol{\mu}_1, \ldots \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$

for $t = 1, \ldots, I$ **do**
    `#pragma omp parallel for (dynamic` **or** T`)`
    for $i = 1, \ldots, N$ **do**
        $\boldsymbol{\tau} = \mathbf{0}$
        $\eta = 0$
        for $j = 1, \ldots, N$ **do**
            if $\|\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)}\|_2^2 \leq R$ **then**
                $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})\mathbf{x}_j^{(t)}$
                $\eta = \eta + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})$
            end if
        end for
        `#pragma omp critical`
        $\mathbf{x}_i^{(t+1)} = \boldsymbol{\tau}/\eta$
    end for
end for
$\{\boldsymbol{\mu}_1, \ldots \boldsymbol{\mu}_k\} \leftarrow$ `reduce_centroids`$(\mathcal{S}^{(I)}, \delta)$
return $\{\boldsymbol{\mu}_1, \ldots \boldsymbol{\mu}_k\}$

CUDA [4]



- Take advantage of **SIMT** model.
- Data stored as a $N \times D$ array $\rightarrow$ 1-dimensional `blockDim`.
- User decides number of threads $T \rightarrow \lceil N/T \rceil$ blocks.
- Exploit both **global** and **shared** memory accesses.

CUDA: Pseudo-code

Input: $\mathcal{S} \subset \mathbb{R}^D$. $I \in \mathbb{N}$. $\sigma, R, \delta \in \mathbb{R}$.
Input: Threads per block: $T \in \mathbb{N}$.
Output: Centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$

1: $B = \lceil N/T \rceil$
2: **for** $t = 1, \dots, I$ **do**
3: `mean_shift«B,T»`$(\mathcal{S}^{(t)}, \sigma, R)$
4: `cudaDeviceSynchronize()`
5: **end for**
6: $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\} \leftarrow$ `reduce_centroids`$(\mathcal{S}^{(I)}, \delta)$
7: **return** $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$

CUDA: *Naive*

Input: $\mathcal{S}^{(t)} \subset \mathbb{R}^D$. $\sigma, R \in \mathbb{R}$.

`tid = (blockIdx.x` $\times$ `blockDim.x) + threadIdx.x`
if tid $<$ N **then**
  $\boldsymbol{\tau} = \mathbf{0}$
  $\eta = 0$
  for $i = 1, \dots, N$ **do**
    if $\|\mathbf{x}_{\mathtt{tid}}^{(t)} - \mathbf{x}_i^{(t)}\|_2^2 \leq R$ **then**
      $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_{\mathtt{tid}}^{(t)} - \mathbf{x}_i^{(t)})\mathbf{x}_i^{(t)}$
      $\eta = \eta + K_\sigma(\mathbf{x}_{\mathtt{tid}}^{(t)} - \mathbf{x}_i^{(t)})$
    end if
  end for
  $\mathbf{x}_{\mathtt{tid}}^{(t+1)} = \boldsymbol{\tau}/\eta$
end if

CUDA: *Shared Memory*

Input: $\mathcal{S}^{(t)} \subset \mathbb{R}^D$. $\sigma, R, \in \mathbb{R}$.

$\omega = \texttt{threadIdx.x}$
$\texttt{tid} = (\texttt{blockIdx.x} \times \texttt{blockDim.x}) + \omega$
$\mathcal{M} \leftarrow \texttt{shared array}[T \times D]$
$\mathcal{V} \leftarrow \texttt{shared array}[T]$
$\boldsymbol{\tau} = \mathbf{0}, \eta = 0$
for $\tau = 1, \ldots, \lceil N/T \rceil$ **do**
  $\gamma = \tau \times T + \omega$
  if $\gamma < N$ **then**
    $\mathcal{M}_\omega^{(\tau)} = \mathbf{x}_\gamma^{(t)}$ **else** $\mathcal{M}_\omega^{(\tau)} = \mathbf{0}$
    $\mathcal{V}_\omega^{(\tau)} = 1$ **else** $\mathcal{V}_\omega^{(\tau)} = 0$
  end if
  `__syncthreads()`
  for $i = 1, \ldots, T$ **do**
    $\mathbf{x}_i^{(t)} \leftarrow \texttt{load\_data\_from\_sm}(\mathcal{M}^{(\tau)})$
    $v_i^{(t)} \leftarrow \texttt{load\_multiplier\_from\_sm}(\mathcal{V}^{(\tau)})$
    if $\|\mathbf{x}_{\texttt{tid}}^{(t)} - \mathbf{x}_i^{(t)}\|_2^2 \leq R$ **then**
      $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_{\texttt{tid}}^{(t)} - \mathbf{x}_i^{(t)})\mathbf{x}_i^{(t)}$
      $\eta = \eta + K_\sigma(\mathbf{x}_{\texttt{tid}}^{(t)} - \mathbf{x}_i^{(t)})v_i^{(t)}$
    end if
  end for
  `__syncthreads()`
end for
if tid $<$ N **then**
  $\mathbf{x}_{\texttt{tid}}^{(t+1)} = \boldsymbol{\tau}/\eta$
end if

Experiments

- CPU: 4C / 8T (Intel i7-860).
- GPU: NVIDIA GTX 980.
- No ϵ-stop $\rightarrow I = 50$ iterations.
- Ten runs for each experiment:

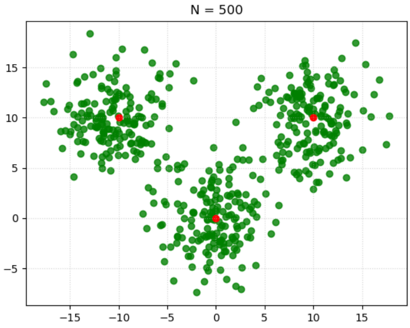
$$\bar{t}_A = \frac{1}{10}\sum_{i=1}^{10} t_A^{(i)} \tag{4}$$

- Speedup of B over A:

$$S = \bar{t}_A / \bar{t}_B \tag{5}$$

- Variance:

$$\sigma^2 \approx S^2 \left[\frac{\sigma_A^2}{\bar{t}_A^2} + \frac{\sigma_B^2}{\bar{t}_B^2} \right] \tag{6}$$



- Four dataset sizes $N = \{500, 1000, 2000, 50000\}$.
- Two dimensionalities $D = \{2, 3\}$.
- Three centroids.

General Behaviour

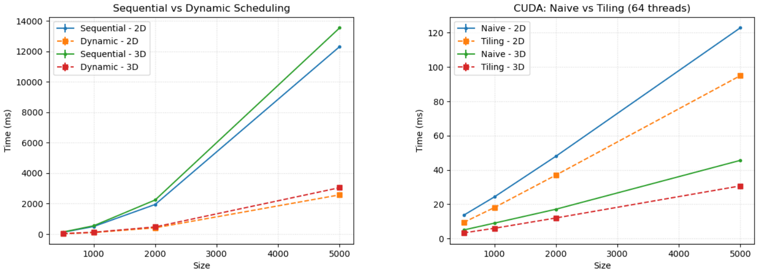


Figure 1: Execution time (*ms*): OpenMP (*left*) (dynamic scheduling) - CUDA (*right*) (with $T = 64$)

OpenMP: Against Sequential

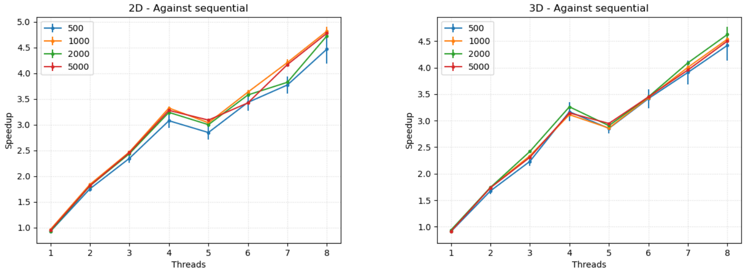


Figure 2: Speedup: Parallel vs Sequential for $2D$ (*left*) and $3D$ (*right*) datasets

- **Maximum** speedup: $S^{max} = 4.82$ for $N = 1000$, $D = 2$ and $T = 8$.
- **Minimum** speedup: $S^{min} = 0.91$ for $N = 500, 1000, 5000$, $D = 3$ and $T = 1$.

OpenMP: Against $T = 1$

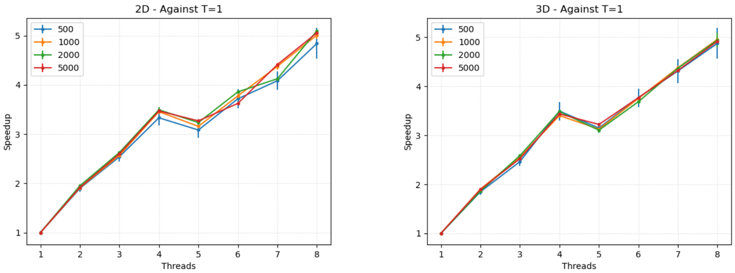


Figure 3: Speedup: Parallel vs $T = 1$ for $2D$ (*left*) and $3D$ (*right*) datasets

- **Maximum** speedup: $S^{max} = 5.10$ for $N = 2000$, $D = 2$ and $T = 8$.
- **Minimum** speedup: $S^{min} = 1$, $\quad \forall N, D$ and $T = 1$.

CUDA: Naive Against Sequential

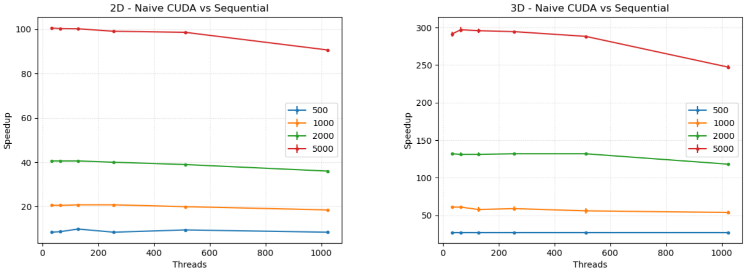


Figure 4: Speedup: Naive vs Sequential for $2D$ (*left*) and $3D$ (*right*) datasets

- **Maximum** speedup: $S^{max} = 297.1$ for $N = 5000$, $D = 3$ and $T = 64$.
- **Minimum** speedup: $S^{min} = 8.46$ for $N = 500$, $D = 2$ and $T = 32, 256, 1024$.

CUDA: Tiling Against Sequential

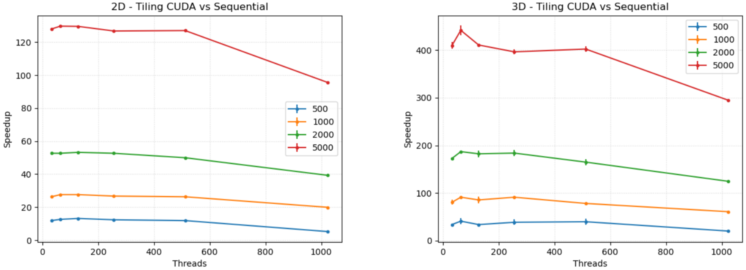


Figure 5: Speedup: Tiling vs Sequential for $2D$ (*left*) and $3D$ (*right*) datasets

- **Maximum** speedup: $S^{max} = 441.32$ for $N = 5000$, $D = 3$ and $T = 64$.
- **Minimum** speedup: $S^{min} = 11.85$ for $N = 500$, $D = 2$ and $T = 32, 512$.

Conclusions

Three implementations of *mean shift*: one sequential and two parallel.

Pros:

- Parallelization methods: OpenMP & CUDA.
- Header-only C++17 libraries, with simple test cases.
- OpenMP: simplest implementation.
- CUDA: obtained a considerable speedup.

Cons:

- OpenMP: speedup not comparable wrt CUDA.
- CUDA: implementation is more difficult.

Code

https://github.com/w00zie/mean_shift

References I

[1] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Transactions on Information Theory*, vol. 21, no. 1, pp. 32–40, 1975. DOI: 10.1109/TIT.1975.1055330.

[2] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, 2002. DOI: 10.1109/34.1000236.

[3] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming,", vol. 5, no. 1, pp. 46–55, Jan. 1998, ISSN: 1070-9924. DOI: 10.1109/99.660313. [Online]. Available: https://doi.org/10.1109/99.660313.

[4] NVIDIA, P. Vingelmann, and F. H. Fitzek, *Cuda, release: 11*, 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.